SDEV 1201 Database Fundamentals

LESSON 8

Query Syntax

Queries are written with the SELECT statement. The simplified syntax for a select statement is:

```
SELECT [ALL | DISTINCT] column_list
[FROM table_name [, table_name2 […, table_name]] -- source of data
          [INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN] -- other sources
[WHERE clause] -- filter
[GROUP BY clause] -- aggregate by
[HAVING clause] -- filter by aggregate
[ORDER BY clause] -- sorts results
```

If it looks complex don’t worry. All the “clauses” listed don’t need to be used when creating a query. However, what clauses you do use MUST be in the order shown above (i.e. HAVING MUST be after GROUP BY for example and not vice versa).

Simple SELECT

```
SELECT
    FirstName
, LastName
, City

FROM Student;
```

Like

The Like operator is used when you use wildcards in your search or perform pattern matching. Wildcards are characters that when used with the Like operator allow the wildcards to represent a character or multiple characters. The Like operator is case sensitive in PostgreSQL.

For example, if we wanted to return all the student's names whose names started with the letter 'D' we would use the % wildcard character. The % wildcard means any character and any number of characters.

```
Select FirstName || ' ' || LastName "Student Name"
From Student
Where FirstName Like 'D%';
```

It is important to use the like operator when using wildcards. The % would be interpreted as a literal '%' if we used = instead of Like.

Like

The _ (underscore) is a single character wildcard. This would be used when you are looking for any one character in your search criteria. If we wanted the students' names whose first name was 3 characters long and started with 'Ja' we could use this query.

```
Select FirstName || ' ' || LastName "Student Name"
From Student
Where FirstName Like 'Ja_';
```

If we had used a % instead of _ we would also return students whose First Name was longer than 3 characters.

Aggregate Functions

Aggregate functions calculate a single value using the data from many records. If you have used function in Excel such as SUM, AVERAGE, COUNT, etc…. then you have used aggregate functions.

```
aggregate_function_name ([ ALL | DISTINCT ] expression)
```

- `AVG(ColumnName)` returns the average of numeric values. `NULL` is ignored.
- `SUM(ColumnName)` returns the sum of a column containing numeric values.
- `MIN(ColumnName)` and `MAX(ColumnName)` returns the minimum & maximum values from a column of numeric, date, or character values.
- `COUNT(*)` returns the number of records that match the `WHERE` criteria.
- `COUNT(ColumnName)` returns the number of non-null values in a given column for records that match the `WHERE` criteria.

String Functions

- LENGTH(*column* | *expression*)
- LEFT(*column* | *expression*, *length*)
- RIGHT(*column* | *expression*, *length*)
- SUBSTRING(*column* | *expression*, *start*, *length*)
- REVERSE(*column* | *expression*)
- UPPER(*column* | *expression*)
- LOWER(*column* | *expression*)
- LTRIM(*column* | *expression*)
- RTRIM(*column* | *expression*)

Date Functions

- `Current_Date` returns the current system date
- `Current_Time` returns the current system time
- `DATE_PART(xx, date1)` returns integer representation of the `xx` of `date1`

`xx` represents field for date portions (see next slide).

Date Function Examples

-- You can also use Fields to get the character representation of dates.

```
Select To_Char (Current_Date, 'DAY');
```

Returns the day of the week (i.e. MONDAY, TUESDAY, etc.). Note that the "Field" is case sensitive: "Day" would return "Monday".

```
Select To_Char (Current_Date, 'MONTH');
```

Returns the month name. Note that the "Field" is case sensitive: "Month" would return "September".

Group By

The GROUP BY clause is used with aggregate functions to provide subtotals. We could easily return the average Mark from the grade table. What if we wanted the average Mark for each course? For each student? For each course for each Student? To do this we would use GROUP BY.

```
SELECT StudentID, AVG(Mark) AS AverageMark
FROM Registration
GROUP BY StudentID
```

calculates the average mark per Student.

We will usually GROUP BY something unique.

Another way to think of the syntax GROUP BY is the 'for each'.

Having

HAVING is like the WHERE clause, except it applies its criteria after GROUP BY. For example:

```
SELECT StudentID, AVG(Mark) as AverageMark
FROM Registration
GROUP BY StudentID
WHERE AVG(Mark) > 80
```

Doesn't work. However, Replacing WHERE with HAVING having does.

```
SELECT StudentID, AVG(Mark) as AverageMark
FROM Registration
GROUP BY StudentID
HAVING AVG(Mark) > 80
```

Joins

For example, if I want to select some fields from the Staff table and other fields from the Position table, which are connected using the PositionID PK/FK

```
select FirstName || ' ' || LastName "Staff Name",
PositionDescription "Position"
from Staff
inner join Position
on Staff.PositionID = Position.PositionID
```

The INNER JOIN syntax tells the server that we are performing an inner join and identifies that PositionID is the field in the 2 tables that relate them together.

Selecting from 3 or more tables

To select from more than 2 tables the syntax would look like:

```
SELECT field1, field2,... FROM table1
INNER JOIN table2
ON table1.joinfield = table2.joinfield
INNER JOIN table3
ON table.joinfield = table.joinfield
INNER JOIN table4
ON table.joinfield = table.joinfield
```

Selecting from 3 or more tables

Select the Student Names, Payment Dates, and Payment Type Descriptions for all the students that have payments:

```
select Student.StudentId, FirstName || ' ' || LastName "Student Name",
PaymentDate, PaymentTypeDescription
from Student
inner join Payment
on Student.StudentID = Payment.StudentID
inner join PaymentType
on Payment.PaymentTypeID = PaymentType.PaymentTypeID
```

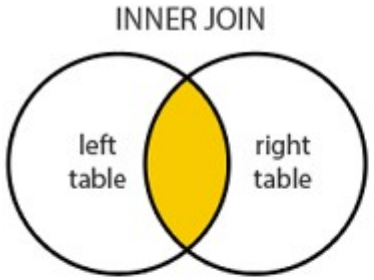
Outer Joins

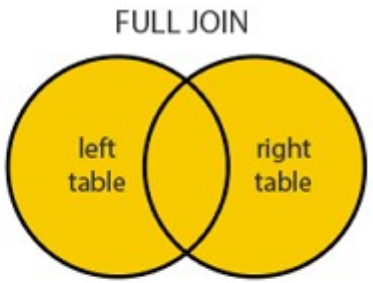
Left Outer Joins and Right Outer Joins both have the same purpose, and you choose which one to use based on the order you list your tables in your select statement.

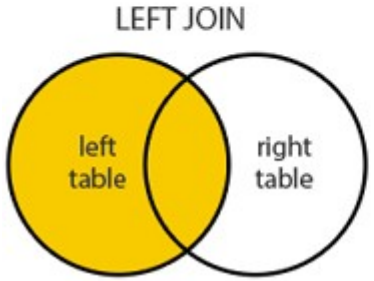
```
select FirstName || ' ' || LastName "Staff Name", PositionDescription
"Position"
from Position
left outer join Staff
on Staff.PositionID = Position.PositionID
```

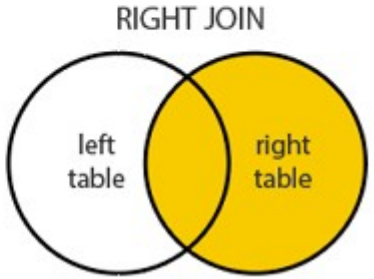
This example states to select ALL the records from the table that is on the LEFT side of the OUTER JOIN statement (Position) and the related records from Staff. The results will include the Assistant Dean position description and a NULL value for staff name.

Types of JOINs









- `INNER OUTER JOIN` returns only records that exist in both tables
- `LEFT OUTER JOIN` returns all records in `table1`, regardless of whether they exist in `table2`
- `RIGHT OUTER JOIN` returns all records in `table2`, regardless of whether they exist in `table1`
- `FULL OUTER JOIN` returns all records that exist in either table (and is seldom used)

Today’s Objective

Today we will be covering:

- Subqueries

Theory and reference material for this presentation is based on the “Advanced Select Statement” document in the “Week 4 – Select Statements” section of Brightspace.

Union

Union

The union operation lets you combine the data retrieved by multiple SELECT statements.

```
SELECT ...
UNION [ALL]
SELECT ...
```

UNION

Let’s write a query to return all the student names and combine it (UNION) with all the staff names.

```
Select FirstName, LastName from Student
UNION
Select FirstName, LastName from Staff
```

This results in 25 records being returned. There are 16 students and 10 staff. Why are we missing one? UNION omits duplicates unless you us UNION ALL. There is both a student and a staff member named “Robert Smith.

You can UNION as many queries together as you need to.

UNION

There are some rules to follow, however:

1. All the queries being combined with UNION must return the same number of columns.

```
Select FirstName, LastName, City
From Student
Union
Select FirstName, LastName
From Staff;
```

ERROR: each UNION query must have the same number of columns

2. The columns being combined must share similar datatypes.

```
Select FirstName, Birthdate
From Student
Union
Select FirstName, LastName
From Staff;
```

ERROR: UNION types date and character varying cannot be matched

UNION

3. Ordering the results must be done after the last query.

```
Select FirstName, LastName
From  Student
Union
Select FirstName, LastName
From Staff
Order By LastName ASC;
```

Works!

4. Column names are defined in the first query.

```
Select FirstName "First Name", LastName "Last Name"
From Student
Union
Select FirstName, LastName
From Staff
Order By "Last Name" ASC;
```

Works!

Class Activity

Please work on the “Union Exercise” found on the Brightspace site in the “Week 4” Section under Activities.

Subqueries

Subqueries

A subquery is a SELECT statement inside of another statement.

It's a full and complete SELECT statement that could execute on its own.

We often refer to the subquery portion (nested inside the other select) as the inner query and the Select statement around it as the Outer Query.

There are nested and correlated subqueries, and in this course we will focus on nested subqueries only.

Subqueries

Let’s look at the following select statement that uses a subquery:

```
Select FirstName, LastName
From Staff
Where PositionID = (Select PositionID
                    From Position
                    Where PositionDescription = 'Dean');
```

Subqueries

The subquery (inner query) executes once and returns one or more values and then the outer query executes using the value(s) returned from the subquery.

Execution example:

```
Select FirstName, LastName
From Staff
Where PositionID = (Select PositionID
                    From Position
                    Where PositionDescription = 'Dean');
```

The inner query (subquery) executes first and returns any values; in this case the subquery returns a value of 1 as the PositionID for ‘Dean’ is 1.

```
Select FirstName, LastName
From Staff
Where PositionID = (1);
```

Subqueries

The outer query then executes using the results of the subquery as criteria for the where clause. The results of the Select executing are:

FirstName	LastName
Hugh	Guy

This is a two-step process where first the subquery returned the PositionID for the PositionDescription of ‘Dean’ and then the Staff Names are selected that had that PositionID. It is important to note that this example used an = in front of the subquery. If our subquery had returned more than one value it would have caused an error when the statement was executed. You can only compare one value to another one value when using an = operator.

Subqueries

What if the subquery returns multiple values?

One of the most common uses of a subquery is to retrieve a list of values that is used as a list of acceptable or non-acceptable criteria by the outer query.

To see all the student names that have made payments, you use the IN operator.

```
Select FirstName, LastName
From Student
Where StudentID In (Select StudentID
                    From Payment);
```

Breaking it down…

1. Subquery executes by first selecting all the studentID's in the payment table.
2. Then the outer query executes using those StudentID's as part of the criteria in the where clause.

Subqueries

What happens when you run the following Select statement?

```
Select FirstName, LastName
From Student
Where StudentID = (Select StudentID
                   From Payment);
```

ERROR: more than one row returned by a subquery used as an expression

Remember, when using = there must be only one value on each side. If the subquery might return more than one value, then you must use the IN operator.

Subqueries

What if I wanted to see all students that have never made a payment?

```
Select FirstName, LastName
From Student
Where StudentID Not In (Select StudentID
                        From Payment);
```

ANY, SOME, and ALL operators

So far, our subqueries have returned values to be used as criteria for an exact match by the outer query.

Where StudentID IN (Select StudentID .....)

What if we want something other than an exact match?

The ANY or SOME (for SQL they are interchangeable) or ALL operators lets you use even more operators (>, <, >=, <=, <>).

ANY/SOME compares against any of the values retrieved by the subquery.

Where Grade > ANY (Select Grade.....)

If the Grade being selected by the outer query is greater than any of the grades returned by the subquery, then the record is returned.

ANY, SOME, and ALL operators

ANY or SOME compares against any of the values:

```
SELECT StudentID
FROM Registration
WHERE Mark > ANY (SELECT Mark FROM Registration)
-- this returns students with marks higher than at least one other
student
```

ALL compares against all of the values:

```
SELECT StudentID
FROM Registration
WHERE Mark > ALL (SELECT Mark FROM Registration)
-- this returns … nothing. Why? How can we return just the top grade?
```

ANY, SOME, and ALL operators

The ALL operator compares against all of the values retrieved by the subquery.

Where Grade > ALL (Select Grade.....)

If the Grade being selected by the outer query is greater than all of the grades returned by the subquery, then the record is returned.

We can use these operators to perform some more complicated select statements. Subqueries can be used in many places where you need to retrieve a set of values for criteria.

ANY, SOME, and ALL operators

Select the City that has the highest number of students in it.

```
Select City
From Student
Group By City
Having Count (StudentID) >= All (Select Count (StudentID)
                                 From Student
                                 Group By City);
```

In this example we used the subquery as criteria in the having clause.

ANY, SOME, and ALL operators

In this example we used the subquery as criteria in the having clause.

Breaking it down:

1. The subquery executes first and returns values (1,1,2,1,8,1,1,1) and the outer query is re-written with those values.

```
Select City
From Student
Group By City
Having Count (StudentID) >= All (1,1,2,1,8,1,1,1);
```

2. The outer query then executes comparing the count of students in each city to see which count(s) are greater/equal than or equal to all the ones in the subquery. The record from the outer query that is equal to the highest value in the subquery is the city with the most students.

Homework

Complete the “Subquery Exercise” which is located in the “Week 4” section of the Brightspace site under Activities. As with the other TSQL exercises, the examples and practice are based on the IQSchool database.

NOTE: Take-home Coding Assigment Advanced SELECT is available on Brightspace. It is **due** on **Friday, October 10, 2025 at 5:00 PM**.

Quiz on Joins and Select Statements will be next Monday on October 6, 2025.

Coding Standards (as found in the course syllabus)

Only those database programming methods, practices, and techniques taught in class, or available from Brightspace notes and/or exercises, will be permissible in SDEV1201 labs and quizzes. Any solution in a lab or quiz submission using non-approved methods, practices, and/or techniques, or code that is not executable in an PostgreSQL environment, will be given a mark of zero.